A Prototype Constraint Deductive Database System based on $HH_{\neg}(\mathcal{C})$

G. Aranda-López[†] S. Nieva[†] F. Sáenz-Pérez[‡] J. Sánchez-Hernández[†]

garanda@fdi.ucm.es nieva@sip.ucm.es fernan@sip.ucm.es jaime@sip.ucm.es

[†]Dept. Sistemas Informáticos y Computación

[‡]Dept. Ingeniería del Software e Inteligencia Artificial

Universidad Complutense de Madrid

Abstract

This paper presents, from a user point of view, a deductive database system based on Hereditary Harrop Formulas with Constraints and Negation, $HH_{\neg}(\mathcal{C})$. The Prolog implementation of this system is based on a fixpoint semantics proposed in a previous work. The answer to a query posed to a database is intended as a constraint of the constraint system associated to $\mathcal{C}$. We have developed several solvers for specific constraint domains, composed of data values and predefined functions and operators, including finite domains, real numbers, Boolean and user-defined enumerated types. They have been implemented by taking advantage of the underlying constraint solvers in SWI-Prolog. In the current version of the system, some improvements regarding the efficiency and the user interface have been introduced. In addition, we have included aggregate functions as `count`, `sum`, `avg` and `min`. We propose to use aggregate functions as components of the constraint language, so that solving of constraints including aggregate functions is delegated to the constraint solver.

1 Introduction

In [3] we presented an extension of Hereditary Harrop formulas with constraints by adding negation to obtain $HH_{\neg}(\mathcal{C})$, a Constraint Deductive Database (CDDB) system, based on a fixpoint semantics as Coral [4]. We stress, as an important benefit of our approach, the ability to formulate hypothetical and universally quantified queries. This resulting language enjoys the expressive power of Datalog but adds constraints and two new logical connectives: implication (to formulate hypothetical queries), and universal quantification (to encapsulate data). We have implemented a prototype as proof of concept for the theoretical framework. Two main components can be distinguished in the implementation of this prototype, that we already presented in [1]. One corresponds to the bottom-up implementation of a fixpoint semantics, which is very close to the theory. The fixpoint is computed using a stratification of the predicates of the database which is obtained from a suitable notion of dependency graph. These graphs include dependencies due to implication and aggregates. The other component corresponds to the implementation of the constraint solvers. The first component is independent of the particular constraint system, i.e., it is parametric on the second component. Then, the known safety results for Datalog (with constraints) [5] are valid in our case, because they rely on the constraint systems.

2 System Description

In this section we introduce the main aspects of our system and how to use it.

2.1 Databases and Queries

As usual in CDDBs, a database is a set of clauses and a query is a goal, whose answer is a constraint. When the system is started, the prompt `HHn(C)>` is displayed. By typing `help`, the available commands are listed. To process a database Δ, the user has to type `run(`*`filename`*`)`, where *`filename`* is the file containing the clauses defining Δ. The execution of this command produces the computation of the fixpoint semantics of Δ following the next computation stages:

1. Check and infer the predicate types.
2. Build the dependency graph of Δ.
3. Compute a stratification s for Δ, if there is any. Otherwise, the system throws an error message and stops.
4. If the previous step succeeds, compute the fixpoint of Δ, $fix(\Delta)$.

The system keeps in memory, while not processing another database, the following information: the just computed $fix(\Delta)$, the stratification s, and the dependency graph of Δ. The fixpoint is stored as a set of pairs (A, C) the atom A can be derived from Δ if C is satisfied, i.e., it captures the semantics of our database [3]. The command `fix` shows the computed fixpoint for the current database.

When a query `G` is posed at the prompt, the system computes, if it exists, a new stratification s' for the set $\Delta \cup \{\texttt{G}\}$. The query can not be computed if there is not such s', and the systems stops. In other case:

- If $s = s'$, the kept fixpoint, computed for Δ, is valid to evaluate `G`.
- If $s \neq s'$, the symbol predicates involved in the computation of `G` can be in a different stratum than when $fix(\Delta)$ was computed. So, the stored fixpoint is not valid now to evaluate `G` and a new fixpoint, $fix(\Delta \cup \{\texttt{query}(\overline{\texttt{X}})\ \texttt{:-}\ \texttt{G}\})$, must be computed, where $\overline{\texttt{X}}$ are the free variables of `G`. The answer will be the constraint `C` stored in the pair `(query(`$\overline{\texttt{X}}$`),C)` of the computed fixpoint.

2.2 Type and Domain Declarations

We have implemented a type checking and inferrer system for $HH_{\neg}(\mathcal{C})$ programs which is able to detect type inconsistencies and lack of type declarations, and to infer types for user queries. A type is known in the context of a set of clauses: Either an atom provides its type (i.e., because of its corresponding predicate type), or a constraint `constr(Dom,C)` provides its type. This type is needed to know the constraint system the constraint belongs to. The system incorporates the predefined data types `bool` (with `true` and `false` as elements), `integer` (for which a finite interval should be provided before being able to use it) and `real`, an infinite data type, whose real numeric range is system-dependent. As well, the user is allowed to define new enumerated data types. A data type declaration is written as:

```
domain(data_type, [constant_1, ...,
                   constant_n]).
```

An n-arity predicate type declaration is written as:

```
type(predicate(type_1, ..., type_n)).
```

Although several solvers can be used together within the same database, they can not be combined for the moment, i.e., constraints of different types cannot be freely mixed to get an heterogeneous compound constraint. Predicates with arguments of different types are restricted to those extensionally defined and that are only intended for informative purposes.

2.3 Constraint Solving

Our constraint systems include "`true`", "`false`", "`=`", "`/=`", "`,`" (conjunction), "`;`" (disjunction), "`not`", and "`ex(X,C)`" (existential quantification).

We have proposed three constraint systems as possible instances of the scheme $HH_{\neg}(\mathcal{C})$: Boolean, Reals, and Finite Domains. The first one includes the type `bool` and adds the universal quantifier (written as `fa(X,C)`, where `X`

is a variable and `C` a constraint). The constraint system Reals includes the type `real` (infinite set of real numeric values), and real constraint operators (`+`, `-`, `*`, ...) and functions (`abs`, `sin`, `exp`, `min`, ...).

Finite Domains represent a family of specific constraint systems ranging over denumerable sets. Enumerated types are included as well as (finite) integer numeric types. Whereas the constraint systems Boolean and Reals have attached predefined types, Finite Domains do not. This system also includes comparison operators (`>`, `>=`, ...), universally quantified constraints (`fa(X,C)`, as above), and the domain constraint `X in Range`, where `Range` is a subset of data values built with both `V1..V2` (where `V1`$\leq$`V2`), which denotes the set of values in the closed interval between `V1` and `V2`, and `R1\/R2`, which denotes the union of ranges. A numeric finite domain also includes constraint operators (`+`, `-`, ...) and constraint functions (`abs`, `min`, ...). Note that relevant primitive functions for each system should be clear from their intended semantics (`+` might not be relevant for Booleans, although it can be used). We allow to use the same symbols to build constraints in different systems; for instance, both `constr(real, X>Y)` and `constr(month, X>Y)` make sense in their respective constraint systems. We rely on the underlying constraint solvers already available in SWI-Prolog [7] for implementing the constraint systems Finite Domains, Boolean and Reals. For certain constraints with finite domain, we map them to constraints in the underlying SWI-Prolog finite domain solver. Before posting to this solver, a constraint is rewritten with the mapped integer values and solve it, and then rewrite it back with the corresponding enumerated values. On the other hand, there are constraints that the underlying solvers cannot directly handle (quantifiers and disjunctions) and they are expanded into search spaces that use only the primitive forms.

2.4 Aggregates

Here we introduce for the first time aggregates as constraint functions as components of concrete constraint systems. Aggregate functions are useful in computing single values from a set of numerical or other-type values. Common predefined functions of relational query languages are `count`, `sum` and `avg`. As our deductive database with constraints scheme gives a natural analogy of the relational calculus, a crucial question concerns with how to extend standard aggregation constructs from the relational model to our scheme in the context of a constraint domain. Attempts to solve this question have been proposed both in geometric constraint databases (see, e.g., Chapter 6 of [2]) and deductive database settings [8]. In attempting to add aggregates to constraint query languages, several obstacles come up:

- Aggregate functions take in a set of values and return a single value. So, as negation, aggregation requires that the involved relation is entirely known.
- The output of queries in those languages are constraints that represent intensional answers, not an explicit set of values.
- In addition, since a constraint answer can represent an infinite set, some aggregate functions, as `count`, may have no sense.

However, we have taken advantage of certain aspects of the semantics of our database system in order to deal with these problems. On the one hand, the stratified design of the fixpoint computation, thought to support negation, is a good frame to incorporate aggregates. On the other hand, aggregates can be represented as functions of a constraint system, then its computation can be relegated to the corresponding constraint solver, taking advantage of its efficiency. As general requirements for computing an aggregate function, we first impose that for the atom A involved in the function the pairs (A, C) of the fixpoint are all already computed. Moreover, C must ground A. Our supported set of aggregates include:

- `count(`*Atom*`)`. *Number of elements*: Returns the number of the ground instances of *Atom* in an interpretation. This function is applied to any domain.

- `sum`(*Atom*, *Var*). *Cumulative sum*: Returns the cumulative sum of the ground instances of *Var* for all the occurrences of *Atom* in an interpretation, where *Atom* includes *Var* as one of its arguments. This function is applied to numerical domains.
- `avg`(*Atom*, *Var*). *Average*: Returns the average of the ground instances of *Var* for all the occurrences of *Atom* in an interpretation, where *Atom* includes *Var* as one of its arguments. This function is applied to numerical domains.
- `min`(*Atom*, *Var*). *Minimum*: Returns the minimum of the ground instances of *Var* for all the occurrences of *Atom* in an interpretation, where *Atom* includes *Var* as one of its arguments. This function can be applied to any domain with a defined ordering between elements in the domain.
- `max`(*Atom*, *Var*). *Maximum*: Analogous to `min`.

Since constraints can now contain aggregate functions, and aggregates include defined predicates, additional considerations must be taken into account in order to compute them. On the one hand, this kind of constraints will introduce negative dependencies in the dependency graph. This is because, similarly as what happens with a clause containing a negated atom in its body, if a clause, defining a predicate p, contains an aggregate over a predicate q, the computation of q must be finished before the computation of p starts. We have implemented aggregate functions over real and finite domains. Inside a constraint of the form `constr(Domain, Exp1 opComp Exp2)`, aggregate expressions may occur into both expressions `Exp1` and `Exp2`, where `opComp` is a comparison operator (=,/=,>, >=,<, =<).

3 The System in Action

In this section we illustrate in detail the different computation stages of the system when working with a concrete example: a database for a bank. This example uses both enumerated datatypes (for clients and branches) and real values (for money amounts). The system is available at `https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems` including a bundle of examples, as `bank.hhc` which illustrates this section.

3.1 Defining the database

First, the following domains are defined:

```
domain(client_dt,
    [smith,brown,mcandrew]).
domain(branch_dt,
    [lon,mad,par]).
```

Where `lon`, `mad` and `par` stand for London, Madrid and Paris respectively. Some type declarations are:

```
type(branch(branch_id,client_dt)).
type(client_id(client_dt,real)).
```

Every argument of the remaining relations has type `real`, as:

```
type(client(real,real,real)).
```

The extensional database is given by the facts:

```
% client_id(Name, Ident)
client_id(smith,1.0).
client_id(brown,2.0).
client_id(mcandrew,3.0).

% client(Ident, Balance, Salary)
client(1.0,2000.0,1200.0).
client(2.0,1000.0,1500.0).
client(3.0,5300.0,3000.0).

% pastDue(Ident, Amount)
pastDue(1.0,3000.0).
pastDue(3.0,100.0).

% mortgageQuote(Ident, Quote)
mortgageQuote(2.0,400.0).
mortgageQuote(3.0,100.0).

% branch(Office, Name)
branch(lon,smith).
branch(mad,brown).
branch(par,mcandrew).
```

As an additional restriction we assume that each client has, at most, one mortgage quote. Next we introduce some views defining the intensional part of the database. The first one captures the clients that have a mortgage: a client has a mortgage if there exists a mortgage quote associated to him.

```
% hasMortgage(Ident)
hasMortgage(I):-
  ex(Q,mortgageQuote(I,Q)).
```

A debtor is a client who has a past due with an amount greater than his balance.

```
% debtor(Ident)
debtor(I):-
  client(I,B,S),
  pastDue(I,A),
  constr(real, A>B).
```

The applicable interest rate to a client is specified by the next relation:

```
% interestRate(Ident, Rate)
interestRate(I,2.0):-
  client(I,B,S),
  constr(real, B<1200.0).
interestRate(I,5.0):-
  client(I,B,S),
  constr(real, B>=1200.0).
```

The next relation specifies that a non-debtor client can be given a new mortgage in two situations. First, if he has no mortgage, a mortgage quote smaller than the 40% of his salary can be given. And, second, if he has a mortgage quote already, then the sum of this quote and the new one has to be smaller than that percentage.

```
% newMortgage(Ident, Quote)
newMortgage(I,Q) :-
  client(I,B,S),
  not(debtor(I)),
  not(hasMortgage(I,Q1)),
  constr(real, Q<=0.4*S).
newMortgage(I,Q) :-
  client(I,B,S),
  not(debtor(I)),
  mortgageQuote(I,Q2),
  constr(real, Q+Q2<=0.4*S).
```

```
% gotMortgage(Ident)
gotMortgage(I):-
  ex(Q,newMortgage(I,Q)).
```

If the client satisfies the requirements to get a new mortgage, then it is possible to apply for a personal credit, whose amount is smaller than 6,000. Otherwise, if such a client does not satisfy that requirements, the amount must be between 6,000 and 20,000. Then we include:

```
% personalCredit(Ident, Amount)
personalCredit(I,A) :-
  gotMortgage(I),
  constr(real,A<6000.0)
  ;
  not(gotMortgage(I)),
  constr(real,(A>=6000.0,A<20000.0)).
```

Moreover it is possible to define a view with the quote and the salary of clients whose mortgage quote is greater than 100 with the next code:

```
% accounting(Ident, Salary, Quote)
accounting(I,S,Q):-
  client(I,B,S),
  mortgageQuote(I,Q),
  constr(real, Q>=100.0).
```

To compute the liquid assets we calculate the cumulative sum of balances of all the clients by including the aggregate `sum` in a constraint expression:

```
% liquid(Amount)
liquid(A) :-
  constr(real,
         A = sum(client(I,B,S),B)).
```

The average salary could be specified by:

```
% avg_salary(Average)
avg_salary(Avg) :-
  constr(real,
         Avg = avg(client(I,B,S),S)).
```

3.2 Fixpoint Computation

In the following, Δ represents the set of clauses previously presented. When this database is

processed, the system builds the dependency graph. Then, from this graph, the stratification algorithm maps:

- Stratum 1 to `client`, `pastDue`, `debtor`, `mortgageQuote`, `interestRate`, `hasMortgage`, `accounting`, `client_id` and `branch`.
- Stratum 2 to `newMortgage`, `gotMortgage`, `liquid` and `avg_salary`.
- Stratum 3 to `personalCredit`.

For instance, `personalCredit` must be clearly in a higher stratum than `gotMortgage` because it negatively depends on `gotMortgage`. In the same way, `avg_salary` is in stratum 2 because it negativelly depends on `client`. Since Δ is stratifiable, the fixpoints $fix_i(\Delta)$ of each stratum i will be computed starting from $i = 1$ up to 3. The computation is based on successive iterations of an immediate consequence operator T_i for each stratum i. The final fixpoint $fix(\Delta)$ is $fix_3(\Delta)$. Next we show these computations for strata:

1. Computation of $fix_1(\Delta)$. The first iteration of T_1 over the empty set obtains in `TI` the pairs defined in the extensional database:

```
(client(1.0, 2000.0, 1200.0), true),
(client(2.0, 1000.0, 1500.0), true),
(client(3.0, 5300.0, 3000.0), true),
(branch(lon, smith), true),
(branch(mad, brown), true),
(branch(par, mcandrew), true),
(client_id(smith, 1.0), true),
(client_id(brown, 2.0), true),
(client_id(mcandrew, 3.0), true)
(pastDue(1.0,3000.0), true),
(pastDue(3.0,100.0, true),
(mortgageQuote(2.0, 400.0), true),
(mortgageQuote(3.0, 100.0), true)
```

   The fixpoint computation of this first stratum requires one more iteration of T_1, that adds the following pairs:

```
(debtor(1.0), true),
(interestRate(2.0, 2.0), true),
(interestRate(X,Y),
   ((X=1.0, Y=5.0); (X=3.0, Y=5.0))),
(accounting(X,Y,Z),
   ((Y=400.0, Z=1500.0, X=2.0);
    (Y=100.0, Z=3000.0, X=3.0))),
(hasMortgage(X), (X=2.0;X=3.0))
```

2. Computation of $fix_2(\Delta)$. The first iteration of T_2 starts with $fix_1(\Delta)$ and adds the following pairs to the set in the first iteration:

```
(newMortgage(X,Y),
  ((Y=<200.0, X=2.0);
   (Y=<1100.0, X=3.0)))
(avg_salary(1900.0), true)
(liquid(8300.0), true)
```

   And, in the second iteration, T_2 adds the pair:

```
(gotMortgage(X), (X=2.0;X=3.0))
```

3. Computation of $fix_3(\Delta)$. The final fixpoint requires only one iteration of T_3 over the currently computed fixpoint $fix_2(\Delta)$ which introduces a pair.

```
(personalCredit(X,Y),
   ((Y>=6000.0,Y<20000.0, X=1.0);
    (Y<6000.0, X=2.0);
    (Y<6000.0, X=3.0)))
```

   This completes the fixpoint computation of $fix_3(\Delta)$. This information, as well as the dependency graph and the stratification is kept for further manipulations of the database.

3.3 Querying

The user can now submit different queries for the database. As a first easy one, we may know if every client belongs to Madrid office:

```
HHn(C)> fa(A,branch(mad,A)).
Answer: false
```

As the query does not imply any change in the dependency graph, it can be solved using the kept fixpoint. A universal quantification over a finite domain is naturally translated into a conjunctive constraint obtained by instantiating the quantified variable with

each element in the domain. In this example, `fa(A,branch(mad,A))` requires to know the constraint associated to `branch(mad,A)`. Exploring the fixpoint we obtain:

```
(mad=lon, A=smith);
(mad=mad, A=brown);
(mad=par, A=mcandrew)
```

which is equivalent to `A=brown`. Then `fa(A,(A=brown))` is transformed into the conjunction:

```
(brown=brown),
(mcandrew=brown),
(smith=brown)
```

which is trivially false.

Dealing with negation, the user may ask for the clients that have not a mortgage:

```
HHn(C)> not(hasMortgage(I)).
Answer: I/=3.0, I/=2.0
```

This query neither changes the stratification and the system uses again the kept fixpoint to solve it. Using aggregate operators we can ask: Assuming that client 2 (Brown) has a past due, what is the `sum` of all the debts in the database?

```
HHn(C)> pastDue(2.0,200.0)=>
          constr(real,
          D=sum(pastDue(I,A),A)).
Answer: D=3300.0
```

Despite of the implication, this query does not change the stratification and so, the current fixpoint can be used. When processing an implication, the maximum stratum of the consequent is computed (1 in this case). Then the system obtains the fixpoint of the database augmented with the antecedent up to this first stratum. Finally, this temporal $fix_1(\Delta \cup \texttt{pastDue}(2.0, 200.0))$ is used to obtain the constraint answer. Note that this computation does not require the recomputation of $fix_3(\Delta)$ because this query does not imply any change in the stratification (the system checks it). To illustrate the opposite situation we can force a change in the stratification by introducing an appropriate dependency in the graph (this is an artificial query for illustrating the situation and it has not a natural reading):

```
HHn(C)> newMortgage(I,R) =>
          interestRate(I,R).
Answer:  R=2.0,I=2.0;
         R=5.0,I=1.0;
         R=5.0,I=3.0
```

Due to the new dependency between `newMortgage` and `interestRate` which this query introduces, the second predicate now jumps to stratum 2. As the stratification has changed, the kept fixpoint cannot be used to solve the query. So, a new temporary clause is added to the current database and a new fixpoint $fix_3(\Delta')$, where:

$$\Delta' = \Delta \cup \{\texttt{query(I,R) :- newMortgage(I,R)=>interestRate(I,R)}\}$$

has to be computed. The answer will be the constraint C, such that this temporary clause is in $fix_3(\Delta')$. After this, the temporal clause and fixpoint are discarded and the previous fixpoint is again restored (there is no need to recompute it).

Acknowledgements: This work has been supported by projects TIN2008-06622-C03-01, S-0505/TIC/0407, S2009TIC-1465, and UCM-BSCH-GR58/08-910502. Also thanks to Jan Wielemaker for providing SWI-Prolog [7] and Markus Triska [6] for both providing its FD library and adding new features we needed.

References

[1] G. Aranda, S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Implementing a Fixpoint Semantics for a Constraint Deductive Database based on Hereditary Harrop Formulas. In *Procedings of the 11th International ACM SIGPLAN Symposium of Principles and Practice of Declarative Programing (PPDP'09)*, pages 117–128. ACM Press, 2009.

[2] G. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.

[3] S. Nieva, F. Sáenz-Pérez, and J. Sánchez. Formalizing a Constraint Deductive Database Language based on Hereditary Harrop Formulas with Negation. In *FLOPS'08, Proceedings*, volume 4989 of *LNCS*, pages 289–304, Ise, Japan, 2008. Springer-Verlag.

[4] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan, and Praveen Seshadri. The coral deductive system. *The VLDB Journal*, 3:161–210, 1994.

[5] P. Z. Revesz. *Introduction to Constraint Databases*. Springer, 2002.

[6] Markus Triska. Generalising constraint solving over finite domains. In *ICLP*, pages 820–821, 2008.

[7] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, 2003.

[8] Carlo Zaniolo. Key constraints and monotonic aggregates in deductive databases. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, pages 109–134, London, UK, 2002. Springer-Verlag.

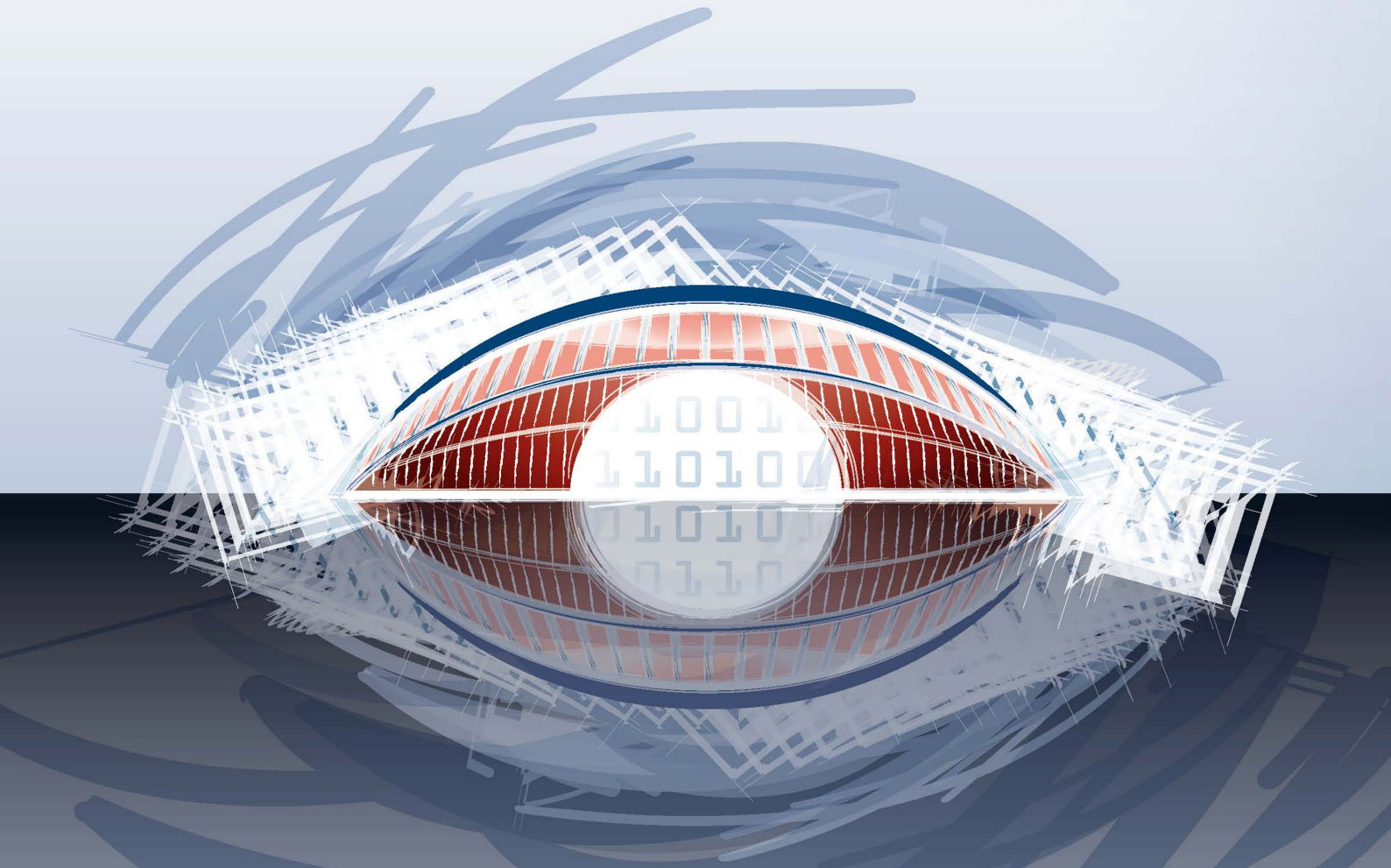
Garceta
grupo editorial

CEDI 2010 VALENCIA

7 A 10 DE SEPTIEMBRE DE 2010

III CONGRESO ESPAÑOL DE INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE VALENCIA



Actas de las X Jornadas sobre Programación y Lenguajes

| PROLE2010 | (SISTEDES)

EDITORES
Víctor M. Gulías, Josep Silva, Alicia Villanueva

Nuevos retos científicos y tecnológicos en Ingeniería Informática

ACTAS DE LAS X JORNADAS SOBRE PROGRAMACIÓN Y LENGUAJES, PROLE2010 (SISTEDES)

EDITORES

Víctor M. Gulías
Josep Silva
Alicia Villanueva

Garceta
grupo editorial

Actas de las X Jornadas sobre Programación y Lenguajes, PROLE2010 (SISTEDES)

Editores: Víctor M. Gulías, Josep Silva, Alicia Villanueva

ISBN: 978-84-92812-55-4

IBERGARCETA PUBLICACIONES, S.L., Madrid, 2010

Edición: 1ª
Impresión: 1ª
Nº de páginas: 290
Formato: 17 x 24
Materia CDU: 004 Ciencia y tecnología de los ordenadores. Informática

Actas del II Workshop de Reconocimiento de Formas y Análisis de Imágenes (AERFAI)

1ª Edición, 1ª Impresión

ISBN: 978-84-92812-55-4

Depósito legal: M-

Maquetación: Los Editores

Coordinación del proyecto: @LIBROTEX

Portada: Estudio Dixi

Impresión y encuadernación:

OI: 18/2010

PRINT HOUSE, S.A.

IMPRESO EN ESPAÑA -PRINTED IN SPAIN

Comité de Programa de PROLE'2010

Presidente Comité: Victor M. Gulías (U. de A Coruña)

Jesús Almendros (U. de Almería)
María Alpuente (U. Politécnica de Valencia)
Puri Arenas (U. Complutense de Madrid)
Miquel Bertran (U. Ramon Llull)
Santiago Escobar (U. Politécnica de Valencia)
Antonio J. Fernández Leiva (U. de Málaga)
Lars-Ake Fredlund (U. Politécnica de Madrid)
María del Mar Gallardo (U. de Málaga)
Paqui Lucio (U. del País Vasco)
Narciso Martí (U. Complutense de Madrid)
Ginés Moreno (U. de Castilla-La Mancha)
Marisa Navarro (U. del País Vasco)
Manuel Núñez (U. Complutense de Madrid)
Fernando Orejas (U. Politécnica de Catalunya)
Yolanda Ortega Mallén (U. Complutense de Madrid)
Francisco Ortín (U. de Oviedo)
Germán Puebla (U. Politécnica de Madrid)
Enric Rodríguez-Carbonell (U. Politécnica de Catalunya)
Julio Rubio (U. de La Rioja)
Jaime Sánchez (U. Complutense de Madrid)
Germán Vidal(U. Politécnica de Valencia)

Comité de Programa de TPF'2010

Presidente Comité: Josep Silva (U. Politécnica de Valencia)

Rafael Caballero (U. Complutense de Madrid)
Laura Castro (U. de A Coruña)
Francisco Gutierrez (U. de Málaga)
José Iborra (U. Politécnica de Valencia)
Salvador Lucas (U. Politécnica de Valencia)
Pablo Nogueira (U. Politécnica de Madrid)
Ricardo Peña (U. Complutense de Madrid)
Mateu Villaret (U. de Girona)

Comité Organizador Local

Coordinadora: Alicia Villanueva (U. Politécnica de Valencia)

Antonio Bella (U. Politécnica de Valencia)
Marco Antonio Feliú (U. Politécnica de Valencia)
Raúl Gutiérrez (U. Politécnica de Valencia)
Daniel Romero (U. Politécnica de Valencia)
Sonia Santiago (U. Politécnica de Valencia)
Salvador Tamarit (U. Politécnica de Valencia)

Comité Permanente

Jesús Almendros (U. de Almería)
María Alpuente (U. Politécnica de Valencia)
Víctor M. Gulías (U. de A Coruña)
Manuel Hermenegildo (U. Politécnica de Madrid)
Juan José Moreno-Navarro (U. Politécnica de Madrid)
Ginés Moreno (U. de Castilla-La Mancha)
Fernando Orejas (U. Politécnica de Cataluña)
Ricardo Peña (U. Complutense de Madrid)
Ernesto Pimentel (U. de Málaga)

Revisores Adicionales

Javier Álvez, Michele Baggi, Antonio Becerra-Terón, Rafael Caballero, David Castro, Sonia Estevez Martín, Henrique Ferreiro, Jose Gaintzarain, Yolanda García Ruiz, Miguel Gómez-Zamalloa, Montserrat Hermo, Luigi Liquori, Alejandro Luna, Miguel Palomino, Jaime Penabad, Adrián Riesco, Juan Rodríguez-Hortalá, Daniel Romero, Francisco P. Romero Chicharro, Fernando Sáenz-Pérez, Salvador Tamarit

Prólogo

Las jornadas sobre PROgramación y LEnguajes (PROLE) se vienen consolidando como un marco propicio de reunión, debate y divulgación para los grupos españoles que investigan en temas relacionados con la programación y los lenguajes de programación.

PROLE'2010 tiene lugar en Valencia durante los días 8 y 10 de Septiembre de 2010, como parte del III Congreso Español de Informática (CEDI'2010), y representa la décima edición de estas jornadas, continuando la tradición de las ediciones anteriores celebradas en Almagro (2001), El Escorial (2002), Alicante (2003), Málaga (2004), Granada (2005), Sitges (2006), Zaragoza (2007), Gijón (2008) y San Sebastián (2009). La presente edición va precedida el día 7 de Septiembre por un taller vinculado a PROLE, el II Taller de Programación Funcional (TPF'2010), que cuenta con su propio Comité de Programa.

En la tradición de los últimos años, PROLE se celebra junto a las Jornadas de Ingeniería del Software y Bases de Datos, auspiciadas por la Sociedad de Ingeniería del Software y Tecnologías de Desarrollo de Software (SISTEDES). Queremos agradecer tanto a los organizadores de CEDI'2010 como a SISTEDES el soporte, infraestructura y apoyo prestados.

Estas actas recopilan tanto los trabajos aceptados para su presentación en PROLE'2010 como TPF'2010. El volumen recopila un total de 30 trabajos que fueron rigurosamente revisados cada uno de ellos por, al menos, 3 miembros de los comités de programa de PROLE/TPF y/o revisores adicionales, a los cuales agradecemos su excelente colaboración y sugerencias para la mejora de los trabajos. Para PROLE'2010 se han seleccionado 26 trabajos que cubren aspectos teóricos y prácticos relativos a la especificación, diseño, implementación, análisis, verificación, validación y aplicación de programas y lenguajes de programación. Por su parte, TPF'2010 ha seleccionado 4 trabajos directamente relacionados con el paradigma de programación funcional.

Además de las restantes actividades vinculadas a CEDI'2010, el programa de PROLE'2010 cuenta con una conferencia invitada que, bajo el título *Property-based testing with Quickcheck*, será impartida por John Hughes, profesor en Chalmers University y CEO de la compañía sueca Quviq. El programa de TPF'2010 cuenta este año con tres seminarios sobre programación funcional impartidos por Carlos Abalde, Gilles Barthe y Pablo Nogueira. A todos ellos queremos agradecer el haber aceptado nuestra invitación.

Por último, queremos agradecer al comité permanente de PROLE la confianza depositada en nosotros para conducir la presente edición de PROLE y TPF, y muy especialmente a nuestros predecesores Ginés Moreno, Ricardo Peña y Paqui Lucio, cuya ayuda y experiencia ha facilitado esta labor.

Septiembre 2010

Víctor M. Gulías
Josep Silva
Alicia Villanueva

Contenido

X JORNADAS SOBRE PROGRAMACIÓN Y LENGUAJES, PROLE2010

1. Conferencia Invitada

2 Taller Programacion Funcional

3 Verificación y Validación

4 Semántica y Análisis

5 Análisis de Programas

6 Programacion Lógica, Fuzzy, Lógico-Funcional y con Restricciones

7 Programación Lógica y Bases de Datos

8 Especificación y Transformación de Modelos y Servicios

9 XML y Web Semántica